AFRL-RI-RS-TR-2016-078

KEVLAR: TRANSITIONING HELIX FROM RESEARCH TO PRACTICE

UNIVERSITY OF VIRGINIA

MARCH 2016

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* MAR 2016	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* FEB 2015 – SEP 2015

4. TITLE AND SUBTITLE KEVLAR: TRANSITIONING HELIX FROM RESEARCH TO PRACTICE	**5a. CONTRACT NUMBER** FA8750-15-2-0054
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 63788F
6. AUTHOR(S) Jack W. Davison, John C. Knight, Michele Co, Jason D. Hiser, Anh Nguyen-Tuong	**5d. PROJECT NUMBER** KEVL
	5e. TASK NUMBER G3
	5f. WORK UNIT NUMBER DE
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Virginia, Office of Sponsored Programs, Charlottesville, VA 22904-4195	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIGA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2016-078

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Security weaknesses in DoD information systems remain a major challenge for system stakeholders. We have advanced technology transition for technology developed under the Helix and PEASOUP projects to protect Air Force systems of interests. The result is an asset that, if widely deployed by the DoD, would enable a high level of confidence in the security of DoD systems, in particular, confidence that certain classes of critical vulnerabilities were no longer subject to possible exploitation. Our technology, called Kevlar, includes key security technologies are protective transformations and targeted recovery. The protective transformations are applied to application binaries before they are deployed. Salient features of Kevlar include applying high-entropy randomization techniques, automated program repairs leveraging highly-optimized virtual machine technology, and developing a novel framework for program analysis, transformation and composition

15. SUBJECT TERMS
KEVLAR, cyber security, binary, lightweight protection, backwards compatibility

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** ANTHONLY R. MACERA
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	40	**19b. TELEPHONE NUMBER** (*Include area code*) N/A

TABLE OF CONTENTS

List of Figures

List of Tables

Kevlar: Transitioning Helix from Research to Practice

1.0 SUMMARY

Security weaknesses in DoD (Department of Defense) information systems remain a major challenge for system stakeholders. We have advanced the transition of technology developed under the Helix and PEASOUP (Preventing Exploits Against Software of Uncertain Provenance) projects to protect Air Force systems of interests. The results are expected to be an asset that, if widely deployed by the DoD, would enable a high level of confidence in the security of DoD systems, in particular, confidence that certain classes of critical vulnerabilities were no longer subject to possible exploitation.

Weaknesses in software code (such as memory overwriting errors, fixed-width integer computation errors, input validation oversights, and format string vulnerabilities) remain common. Exploiting these weaknesses, attackers are able to hijack an application's intended control flow to violate security policies (exfiltrating secret data, allowing remote access, bypassing authentication, or eliminating services). To mitigate and defend against attacks that seek to exploit such weaknesses, we have developed the Helix architecture. Helix represents the culmination of over 10 years of R&D with support from Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF), the Army and the Air Force, and ongoing support from the Intelligence Advanced Research Projects Agency (IARPA).

We have leveraged the opportunity to take the Helix architecture one step closer to deployment in real systems by enhancing it to be a completely automatic system for securing applications against attack by well-funded, determined malicious adversaries. Helix/Kevlar armors binary programs and protects them from attacks, which could arise from the inevitable vulnerabilities that remain after deployment. Source code of the application to be protected is not required nor is any other development artifacts. These features make Helix/Kevlar of particular value for software systems that have to be used but for which no development information is available, or for which significant portions of the system include handwritten assembly code and special-purpose libraries.

The key security technologies used by Helix/Kevlar are protective transformations and targeted recovery. The protective transformations are applied to application binaries before they are deployed. Conceptually, these transformations are tailor-made, lightweight "armor" that prevent an attacker from exploiting residual vulnerabilities in a wide variety of classes. Helix/Kevlar uses novel, fine-grained, high-entropy diversification transformations to prevent an attacker from successfully exploiting vulnerabilities. To prevent attacks from causing the system to act in undesirable ways, such as crashing or performing unintended actions, Helix/Kevlar also provides custom-made, application-specific remediation strategies that may be invoked in the event of an attack.

An important development for the project was the integration of our static binary rewriting technology into Helix/Kevlar. With the addition of the binary rewriting technology (called Zipr) into Helix/Kevlar, we can instantiate protections statically using Zipr or dynamically using Strata. Zipr is appropriate for resource-constrained devices (e.g. embedded systems, Internet of

Things) and applications that include virtual machines such as Java and Javascript, while Strata is appropriate for systems where a moving target defense is appropriate.

Helix/Kevlar has several major strengths: (a) it is applied to binaries and does not depend on particular languages, compilers, or libraries, (b) it is complementary to other security techniques including inspection, static analysis and testing, (c) it requires no changes to the software development process, and (d) preliminary performance measurements show that the armoring provided by Helix/Kevlar is lightweight incurring modest run-time performance overhead of around 10% when dynamic translation is used, and less than 5% when static rewriting is used.

Another notable achievement of the project was that we retargeted Strata to Windows (64-bit). With this addition, Helix/Kevlar can now be applied to Windows applications.

In summary, major accomplishments of the project include:

- Retargeted Strata, Helix/Kevlar's dynamic translator, to 64-bit Windows,
- Integrated Zipr, an efficient static binary rewriter into Helix/Kevlar,
- File a U.S. Patent on the Zipr technology (Title: System, Method and Computer Readable Medium for Space-Efficient Binary Rewriting),
- Demonstrated the ability to apply Helix/Kevlar transforms to the core libraries of the Java VM,
- Collaborated with Northrop Grumman to do an evaluation of the effectiveness of Helix/Kevlar in protecting applications,
- Accepted paper at the 10th IET System Safety and Cyber-Security Conference describing Helix/Kevlar's protection of binary programs,
- Accepted paper at the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks describing a new taint inference technique for defeating web application attacks, and
- Accepted paper at the 11th Annual Cyber and Information Security Research Conference describing how Helix/Kevlar can be used to defeat blind ROP attacks.

2.0 INTRODUCTION

Security weaknesses in DoD information systems remain a major challenge for system stakeholders. To mitigate and defend against attacks that seek to exploit such weaknesses, we have developed the Helix architecture. Helix represents the culmination of over 10 years of Research and Development (with support from DARPA, the National Science Foundation, the Army and the Air Force, and IARPA). Salient features of Helix/Kevlar include developing high-entropy randomization techniques, automated program repairs, leveraging highly-optimized virtual machine technology, and in general, developing a novel framework for program analysis, transformation and composition. We propose to transition technology developed under the Helix and PEASOUP projects to protect Air Force systems of interests. We expect the result to be an asset that, if widely deployed by the DoD, would enable a high level of confidence in the security of DoD systems, in particular, confidence that certain classes of critical vulnerabilities were no longer subject to possible exploitation.

The next two sections describe the Helix/Kevlar architecture and our plans to transition this technology so that it can be used to protect current and future Air Force systems. The major component of this effort is to develop Helix/Kevlar, a robust easy to use tool for applying the Helix technology to real systems.

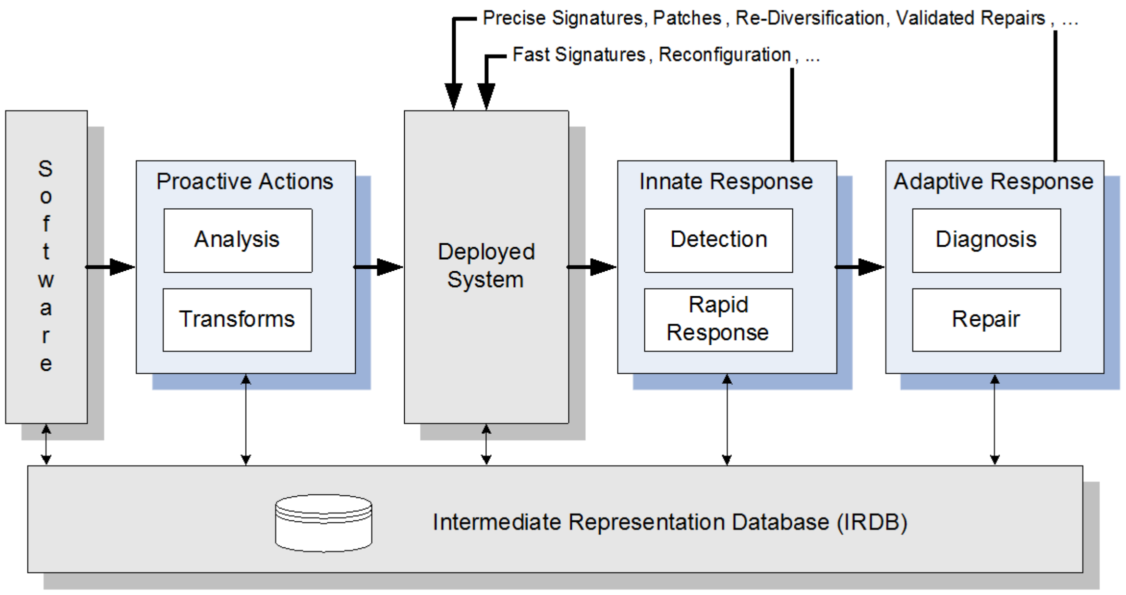
3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Helix

A fundamental problem with current defenses is that they do not redress the asymmetry between attackers and defenders, changing the target system only slowly and reactively in response to attacks. Even approaches that incorporate intrusion detection and tolerance have proven ineffective against determined and well-funded attackers who have at their disposal a growing arsenal of evasive, stealthy, adaptive, polymorphic and metamorphic attacks. To cope with such sophisticated attacks, the Helix architecture uses a combination of defense mechanisms that is both highly effective and metamorphic, i.e., a *high-entropy metamorphic shield*, that presents attackers with a continuously changing attack surface.

Figure 1 provides a high-level conceptual overview of the Helix architecture. An application running in Helix is treated in a holistic way, with information being shared across development, deployment, execution, and response phases in ways that are not possible with traditional architectures.

Instead of viewing the standard tool chain as just a series of steps to transform an application from source code to executable form, we take a more comprehensive view in which program metadata can be deposited in the Intermediate Representation Database (IRDB), and subsequently manipulated and enhanced at all phases of a program's lifecycle, to enable the development of novel and accurate security protection algorithms. Starting with applications in source or binary form as input, Helix proactively analyzes and transforms applications to augment them with self-sensing and self-protection capabilities. Helix enables innate and adaptive actions in response or in anticipation to attacks by running applications under control of Strata, a lightweight virtual machine known as a software dynamic translator (SDT). Strata provides the ability to rewrite application code on-demand for deploying security protections and/or dynamically shifting the attack surface of applications.



Precise Signatures, Patches, Re-Diversification, Validated Repairs, ...
Fast Signatures, Reconfiguration, ...
Software
Proactive Actions
Analysis
Transforms
Deployed System
Innate Response
Detection
Rapid Response
Adaptive Response
Diagnosis
Repair
Intermediate Representation Database (IRDB)

Figure 1: High-level conceptual overview of the Helix architecture

Instead of viewing the standard tool chain as just a series of steps to transform an application from source code to executable form, we take a more comprehensive view in which program metadata can be deposited in the Intermediate Representation Database (IRDB), and subsequently manipulated and enhanced at all phases of a program's lifecycle, to enable the development of novel and accurate security protection algorithms. Starting with applications in source or binary form as input, Helix proactively analyzes and transforms applications to augment them with self-sensing and self-protection capabilities. Helix enables innate and adaptive actions in response or in anticipation to attacks by running applications under control of Strata, a lightweight virtual machine known as a software dynamic translator (SDT). Strata provides the ability to rewrite application code on-demand for deploying security protections and/or dynamically shifting the attack surface of applications.

Helix is a multi-faceted research vision that has lead to many key results. However, as a research project, some ideas are more suited to current, real-world use than others. To transition the best, most deployable of these ideas to practice, from Technical Readiness Level 5 (TRL-5: testing of integrated technology components in representative environment) to Technical Readiness Level 6 (TRL-6: prototype implementation on full-scale realistic systems), we introduce Helix/Kevlar. Kevlar directly leverages the following capabilities from the Helix project:

- The concept of analyzing and storing meta information regarding software in the Intermediate Representation Database is key to enabling various security transformations.

- High precision static and binary analysis of binaries.
- Helix incorporates several novel high-entropy randomization techniques.
- Helix significantly advanced the use of fast dynamic binary rewriting techniques for armoring binaries without requiring the availability of source code.
- Helix leverages the Strata virtual machine technology for transparently augmenting binaries with self-sensing, self-diversification, self-protection and self-repair capabilities.

Overall, Helix provides the intellectual framework for quickly developing and fielding new security transformations. The next section describes how Helix/Kevlar will exploit Helix-developed capabilities in order to begin an effective transition from research to practice.

3.2 Helix/Kevlar Architecture

Helix/Kevlar is a completely automatic system for securing applications against attack by well-funded, determined malicious adversaries. It armors binary programs and protects them from attacks which could arise from the inevitable vulnerabilities that remain after deployment. The source code is not required nor are any other development artifacts (such as object files, debugging information, linker maps, etc.) Enabling the rapid development of security transformations and enabling their safe composition are hallmarks of the Helix/Kevlar system. Helix/Kevlar consists of two phases: (1) an offline phase in which Helix/Kevlar performs deep analyses on binaries and records results in an intermediate representation database (called the IRDB). Helix/Kevlar then uses the database to generate and vet sprockets, i.e. specifications for security transformations; and (2), an online phase in which these sprocket specifications are applied. They can be applied using Strata, a state-of-the-art dynamic binary rewriter [25, 34]. In addition, in this project, we developed Zipr, a highly efficient static binary rewriter [8]. Zipr provides the ability to apply Helix/Kevlar protections to systems that use self-modifying code (e.g., just-in-time compilers such as Java).

Figure 2 shows the high-level architecture of the off-line or redeployment portion of Helix/Kevlar. Helix/Kevlar consists of a static analyzer, called STARS [10], that disassembles x86 binaries, performs extensive static analysis of the binary, and then stores the results of the analysis along with the binary persistent in the IRDB.

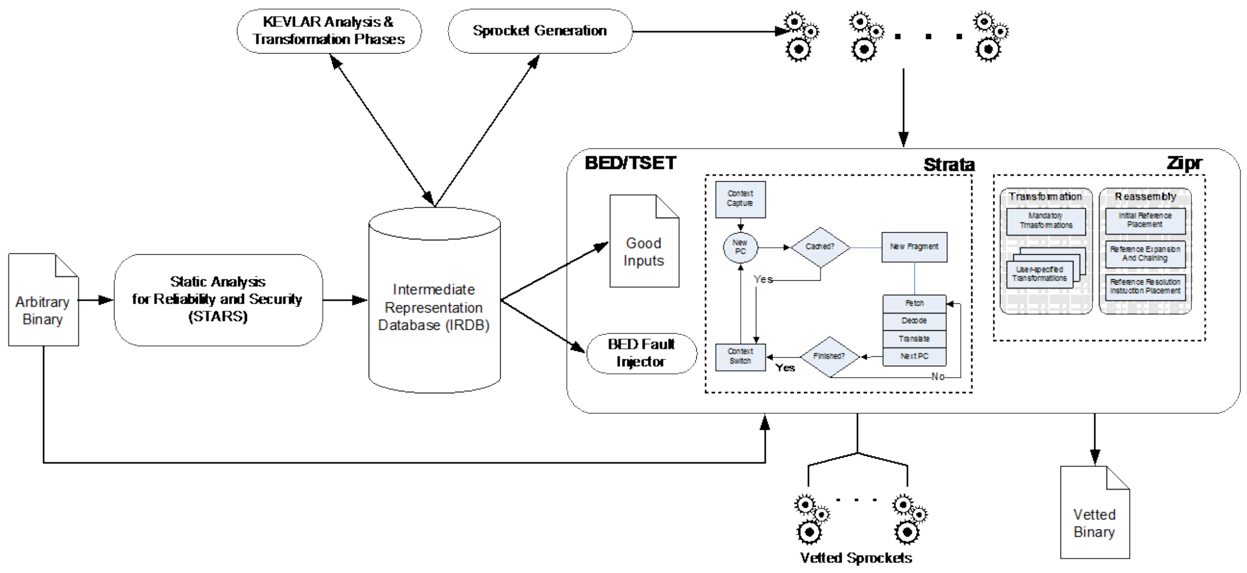

KEVLAR Analysis & Transformation Phases
Sprocket Generation
Arbitrary Binary
Static Analysis for Reliability and Security (STARS)
Intermediate Representation Database (IRDB)
BED/TSET
Good Inputs
BED Fault Injector
Strata
Context Capture
New PC
Cached?
New Fragment
Yes
Fetch
Decode
Translate
Next PC
Context Switch
Finished?
Yes
No
Zipr
Transformation
Mandatory Tnasformations
User-specified Transformations
Reassembly
Initial Reference Placement
Reference Expansion And Chaining
Reference Resolution Instruction Placement
Vetted Sprockets
Vetted Binary

Figure 2: Helix/Kevlar Architecture: Offline generation of Sprockets programs.

A Helix/Kevlar transformation phase uses information in the IRDB to create new versions of a binary, called variants, where various armoring transformations and remediation policies have been applied. A novel aspect of Helix/Kevlar is that, rather than statically rewrite the binary, Helix/Kevlar produces programs, called Sprockets, that are used by the Helix/Kevlar rewriters to transform the original binary into the corresponding variant at run time.

To ensure that the variants produced by Helix/Kevlar run appropriately, they are then "vetted" by a tool called BED (Behavior Equivalence Detection) and TSET (Test Suite Evaluation Technology). BED runs each variant using a regression test suite to ensure that the variant produces the same output as the original binary while TSET seeks to measure confidence levels of the results reported by BED. In addition, BED uses a fault injector to inject faults into the application to determine the effectiveness of the Helix/Kevlar remediation policies.

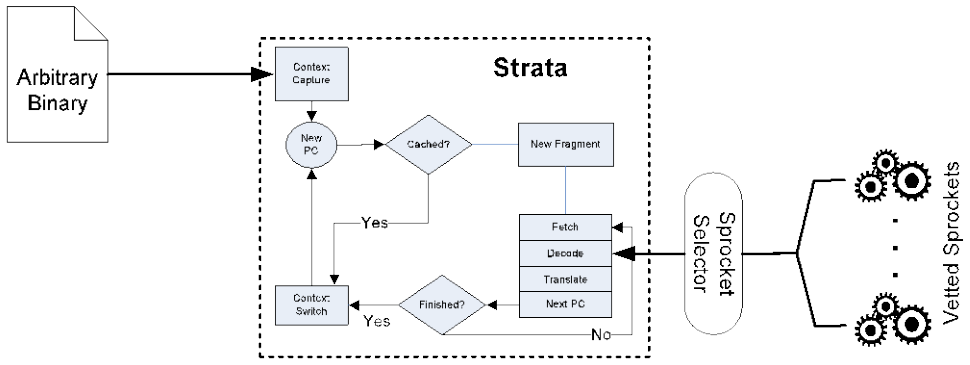


Figure 3: Helix/Kevlar Architecture: Online Selection of Sprocket programs.

Figure 3 shows a deployed binary that is protected by Helix/Kevlar. In this diagram, the vetted Sprocket programs are applied to the binary by the software dynamic translator, Strata. Helix/Kevlar has the ability to dynamically select from the set of Sprocket programs to effect temporal change in the protections that are applied. Such changes could be triggered periodically or because an attack has been sensed and remediated and it is desired to add additional protections or to apply different remediation policies.

We highlight several major analysis and security transformations supported by Helix/Kevlar. Each transformation can be used in isolation, or composed with other transformations for added protection.

3.2.1 Intermediate Representation Database (IRDB)

To facilitate multiple simultaneous transformations to a program, Helix/Kevlar uses an intermediate representation (IR) held in a database, which we term the IR database (IRDB).

The IRDB is similar to the IR for a traditional compiler. It contains information about the program, such as the instructions which make up the program, their addresses, their control and data flow, etc. Furthermore, it contains information about each function including the stack layout, entry points, exit points, etc., the global data layout of the program, targets of indirect branches, etc. Program information is added to the IRDB by various tools including a binary static analyzer called STARS that is discussed in the next section (Section).

Unlike a traditional IR for a compiler, the IR in the database is not guaranteed to be 100% accurate. We realistically assume that information such as perfectly accurate disassembly of the program is not available. We make this assumption to facilitate binary analysis and transformation where such information is rarely available. Issues of imperfect analysis can be compounded if different analyses disagree on information. For example, we use both STARS and the Linux utility `objdump` to populate the list of instructions in the IRDB. The two tools typically agree on instruction start addresses, but occasionally they disagree. The IRDB facilitates the use of conflicting information by supporting conflicting information with a confidence metric.

Another key feature of the IRDB is the ability to “clone” a program. A cloned program is identical to the program it was cloned from, except with a new name, and extra information to track the creation of clones. The primary purpose of a clone is to facilitate programmatic experimentation. For example, suppose that we wish to determine which remediation technique would be effective for a given program. We might choose to clone the program, then instrument the program with the remediation technique for testing.

The clone feature has one other primary purpose: namely we need a “before” and “after” version of the program to support automatic generation of the Sprockets needed to execute the modified program. By tracing a cloned program’s hierarchy back to the untransformed program, we can successfully generate Sprockets to represent the changes between the original program and the transformed program. By examining these differences, automatic generation of the Sprockets is fast, efficient and reliable.

Helix/Kevlar's IRDB is implemented using PostgreSQL. Measurements show that it is fast and efficient.

3.2.2 Static Analysis for Reliability and Security (STARS)

A key component of Helix/Kevlar is STARS (STatic Analyzer for Reliability and Security). It was developed to determine certain security properties of a binary program [10]. STARS is implemented as a plug-in to the popular IDA Pro disassembler [9]. The static analyzer currently operates on Linux/x86 binaries, although it can be targeted to any platform that is targeted by IDA Pro. Currently, IDA Pro targets more than 40 processors and operating platforms.

A key function of STARS in Helix/Kevlar is the identification of the instructions of the application. As discussed by Debray and Andrews, precisely disassembling a binary is, in general, not a solvable problem [24]. In practice, STARS rarely misidentifies data as code. As noted by Debray and Andrews, such misidentifications would be disastrous for a static code rewriter. Because Helix/Kevlar uses software dynamic translation to transform code, Helix/Kevlar is able to tolerate any inaccuracies—we will never rewrite data as code as the rewrite process occurs during the fetch/execute/translate phase of the dynamic translator. That is, only code that should be executed is processed.

The static analyzer analyzes the control flow and data flow of the entire program binary. The analyzer builds a fully pruned SSA (Static Single Assignment) form representation of the program and performs numerous data flow analyses on this representation [4,31]. The data flow analyses include a simplified type system, in which registers and stack locations are typed as being data pointers, integers, floating-point values, strings, or code pointers.

All information determined by STARS is recorded in the IRDB. Later analysis and transformation phases of Helix/Kevlar use this information. During these phases, as additional or more accurate information becomes known, information in the IRDB is updated.

3.2.3 Sprocket Execution Engines

In Helix/Kevlar, transformations to the binary are either applied statically or dynamically. Each approach has advantages. Static rewriting typically has a smaller memory footprint and lower run-time overhead. Dynamic rewriting supports a moving target defense by constantly transforming the application. It also permits a single binary to be deployed with transformations applied dynamically to create an ever-changing attack surface—the metamorphic shield. While the overhead of dynamic rewriting is reasonable, for resource-constrained devices, static rewriting may be preferred. The rewrites to be applied are specified by sprocket program expressed using the Sprocket Program Rewriting Interface (SPRI).

SPRI and Sprocket Generation

SPRI defines simple rewriting rules that come in two forms. The first form, the redirect form, transfers control to a specified target address (lines 1 and 3 in Figure 4). The second form, the instruction definition form, indicates that there is an instruction at a particular location (line 2). The net effect of applying the SPRI rules shown in Figure 4 is to rewrite the instruction `sub esp, 20` instruction at address `0x8000` to be `sub esp, 40`. The stack layout transformation (described in Section 3.3.2) uses such rules to transform stack frame allocations.

Together these two types of rules provide the foundation for building a wide range of Sprocket programs. The example shown illustrates the equivalent of a small patch that modifies only 1 instruction. At the other end of the scale, transformations such as ILX (instruction location transformation, described in Section 3.3.1) seek to rewrite all instructions in a binary.

```
Original Program Fragment:
(a) 0x8000  sub esp,20

Rewrite rule:
(1) 0x8000 -> 0xFF00
(2) 0xFFF0 ** sub esp,40
(3) 0xFF01 -> 0x8001
```

Figure 4: Sprocket rewrite rule to change stack frame allocation. For exposition purposes, all instructions are 1-byte long.

Despite its conceptual simplicity, manually writing Sprockets in SPRI would be a tedious and error-prone process. Instead, Sprocket developers apply their transformations using a high-level C/C++ API (application program interface) to manage the creation and deletion of program variants, and to manipulate program state, e.g. to insert, delete, or replace instructions and re-route control flow. The API transparently interacts with the IRDB to commit any changes.

With this architecture, the composition of Sprockets is naturally performed by chaining together transformations: one Sprocket encodes its transformation in the IRDB, the next Sprocket then takes as input the new database state, and then effects its own transformations. Helix/Kevlar then automatically generates SPRI rules for any program variants by essentially performing a “smart diff” between the IRDB representations of a variant and the original binary.

Once the SPRI rules are generated, Helix/Kevlar can use either Strata or Zipr to instantiate the transformations.

Strata

Figure 3 shows the dynamic translation of a binary using Strata to apply the transformations specified by the Sprockets. While we use Strata as our underlying SDT infrastructure, we note that Sprockets could be similarly implemented via any SDT tool [3, 17, 19, 25, 26, 29].

Strata dynamically loads an application and mediates application execution by examining and translating an application's instructions before they execute on the host CPU. Strata operates as a co-routine with the application that it is protecting. Translated application instructions are held in a managed cache called a fragment cache. Strata is first entered by capturing and saving the application context (e.g., program counter (PC), condition codes, registers, etc.) Following context capture, Strata processes the next application instruction. If a translation for this instruction has been previously cached, Strata transfers control to the cached translated instructions.

If there is no cached translation for the next application instruction, Strata allocates storage in the fragment cache for a new fragment of translated instructions. Strata then populates the fragment by fetching, decoding, and translating application instructions one-by-one until an end-of-fragment condition is met (e.g., an indirect branch). As the application executes under Strata's control, more and more of the application's working set of instructions materialize in the fragment cache.

Implementation of Sprockets requires several simple extensions to a typical software dynamic translator. First, we must modify Strata startup code to read the SPRI rewrite rules (not pictured). Next, Strata's instruction fetching mechanism is overridden to first check, then read from SPRI rewrite rules as appropriate. Lastly, the next-PC operation is modified to obey any redirection rules that are specified.

Finally, we must take steps to protect Strata itself. To thwart a compromised application from overwriting Strata's own code or data, we use standard hardware memory protection mechanisms. When executing the untrusted application code, Strata turns off read, write, and execute permission on the pages of memory it uses, leaving only execute (but not write) permission on the code cache. Strata also watches for attempts by the application to change these permissions. Previous work has shown this technique to be effective and that it costs very little [13].

Zipr

As noted previously, dynamic rewriting and software dynamic translation are not appropriate in all cases. One type of program that causes dynamic rewriting techniques to have high overhead is a program that generates code dynamically, such as Java JIT Compiler or JavaScript engine. Dynamic translation must flush cached code whenever new code is generated, which causes the translation system to do extra work. Typically slowdowns might be at least 3 times slower than the untranslated program. A new addition to the Helix/Kevlar toolchain during the performance period was Zipr, a highly efficient static binary rewriter.

Zipr addresses many of the problems suffered by existing static binary rewriters such as high space overhead (as much as 2X), the inability to translate arbitrarily compiled programs (such as code compiled to be position independent), high runtime overhead, and requiring additional compiler information which may not be available [2, 6, 15, 23, 28, 30, 32, 33, 37].

As part of this contract, we created a new type of static rewriter. Our breakthrough technology, called Zipr, is capable of re-using existing code segments and disambiguating any data that may reside within the code. Using deep static analysis provided by STARS and the analyses provided to build the IRDB, our prototype vastly outperforms existing static rewriting. Zipr can transform arbitrary binaries, compiled by any compiler, and has modest runtime and memory overheads. The following paragraphs provide additional details.

Our technique uses *pinned addresses*, locations of instructions in the original program/library that may be targeted indirectly at runtime. Addresses of units of data are always pinned. On the other hand, the address, *a*, of an instruction, *i*, in the original program is pinned if the original program calculates dynamic program control references to *a* at runtime. In this case, *a* will be stored as the pinned address value of *i*. In other words, pinned address analysis depends directly on correct calculation of indirect control flow.

Addresses of instructions may be pinned for a number of reasons. Most commonly, however, address are pinned because they are the targets of indirect branches (IB). IB targets (IBTs) appear in jump tables, immediately after call instructions, the beginning of functions, etc. Just because program control reaches an instruction indirectly does not mean that it's address must be pinned. There are cases where the program's behavior with respect to an IBT can be analyzed and modeled statically.

For our rewriting methodology to operate correctly it is *not* necessary to determine the set of possible targets for every particular indirect branch instruction. Our technique relies only on the fact that *P*, the set of all pinned addresses, contains *at least* all the addresses of IBTs in the original program. In other words, we rely on the creation of *P* such that $B \subseteq P$ where *B* is the set containing the addresses of every IBT from the original program.

It is possible to calculate *P* naively by making the address of every instruction of the original program a member. This assignment clearly satisfies the requirement. As explained when describing reassembly such an assignment does not give the reassembly technique the flexibility to re-place instructions. Moreover, it does not allow for the creation of an *efficient* rewritten binary program.

Ideally $B=P$. As $|P-B|$ grows, our method generates an increasingly less space-efficient rewritten binary. Therefore, our algorithm leverages a set of heuristics that analyze the original program's CFG to select pinned addresses. Again, it is imperative that our technique be conservative; missing a pinned address will cause our rewriting algorithm to generate a transformed binary that does not operate correctly.

For a more detailed description of the algorithms used to identify pinned addresses of instructions and data, see Hiser et al. [11] and Zhang et al. [36]. For binaries generated by GCC, target compiler of Helix/Kevlar, our prototype implementation, we are able to handle very complex programs including libraries such as `glibc`. Empirical evidence suggests that Helix/Kevlar works for programs generated by LLVM as well.

Pinned addresses of instructions play an important role in reassembly. Throughout the rewriting algorithm, a pinned address, *a*, of an instruction in the original program corresponds to exactly one instruction, *i*. IR Construction assigns the original correspondence between *a* and *i*. During the Transformation phase, one or more transformations will change *i* to *i*' and *a* will still correspond to *i*'. For the modified program to function according to the semantics of the original program, as subsequently modified through user-specified transformations, when the transformed program's program counter (PC) reaches address *a*, instruction *i*' must be executed. The Reassembly phase (described later) maintains this condition.

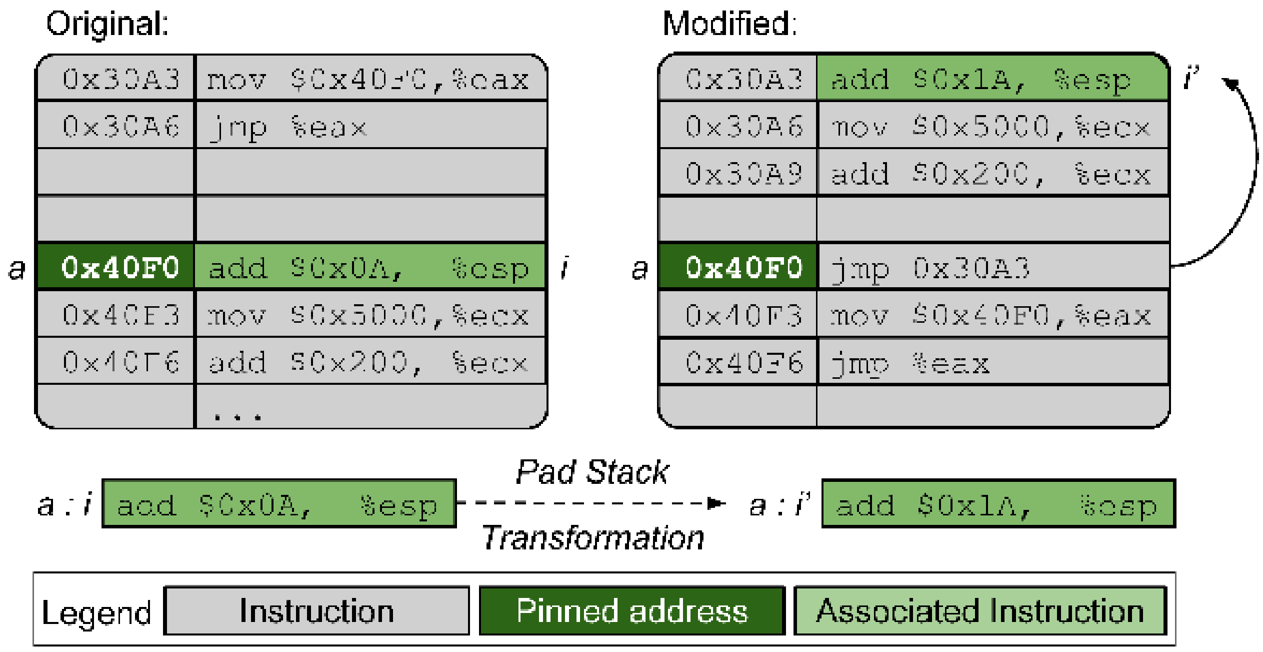


Figure 5: Pinned addresses, transformations and references.

Figure 5 shows an example of this process. Instruction *i* is associated with pinned address *a*. The *Pad Stack* transformation modifies *i* so it allocates a larger stack. The modified instruction, *i*', is still associated with *a*. When *i*' is eventually placed at address `0x30A3` in the modified program, the reference at *a* is updated appropriately.

Zipr's Transformation phase modifies the original program's IR. User-specified transforms are optional transformations that modify or add/remove functionality to/from the original program. Mandatory transforms make it possible for the user-specified transforms to modify the original program's IR without regard for the details of the specific target platform.

Mandatory transformations in the Transformation phase produce a modified IR that makes it possible for the reassembly algorithm to place recreated instructions arbitrarily in the modified program's address space.

Mandatory transformations most commonly address issues with the target platform and its ISA. For example, many x86 instructions can use PC-relative addressing. The jump instruction is one such instruction.

Assume instruction i_1 transfers control to i_2 with a jump. On an x86, i_1 is a jump to a_{i_2}, the address of i_2. However, a_{i_2} is encoded in i_1 relative to a_{i_1}. To be able to relocate instructions, relationships like these that rely on the instructions' addresses in the original program have to be translated into logical links. Fortunately, the IR is built using logical connections among instructions. Returning to the example, the IR links i_1 to i_2, not a_{i_2}. Memory operations (loads and stores) may also be PC-relative.

Unless this situation is handled, PC-relative instructions' placement in the modified program at different addresses will cause an error during execution of the modified program. Each target platform's ISA is different and our method's modular approach makes it possible for the user to apply as many mandatory transformations as necessary to accommodate the target platform. Helix/Kevlar includes all the required mandatory transformations for the x86 and x86-64 platform.

Once all the mandatory transformations are applied, our technique applies any user-specified transformations. These are transformations that the user implements that will modify the original program. As mentioned earlier, there are many ways a user could modify the original program to improve its security, reliability and dependability.

Instead of forcing the user to choose from a set of predefined transformations, Helix/Kevlar provides the user an API to develop their own. The API allows the user to iterate through the functions and instructions of the original program. Users can change (modify or replace) or remove instructions. They can even add new instructions or specify how to link in pre-compiled program code and execute functions therein.

At the heart of our approach's novel reassembly technique is an algorithm that carefully reassembles the modified IR into a series of instructions and units of data which are then assigned a location in the modified program's address space.

The process begins by creating references in the modified program at the pinned addresses from the original program. These references target an pinned address' associated instruction or unit of data, as explained previously. The targets of those references (and their fall-through instructions) are placed arbitrarily in the remaining free space and the references are marked as resolved. In the process of resolving the initial set of references, new unresolved references may be introduced. The targets of those references are again placed arbitrarily in the remaining free space and the references are resolved. The process continues until there are no more unresolved references.

Figure 6 illustrates the internal state of the algorithm as it reassembles a program. The following subsections explain the reassembly algorithm in detail.

At the outset, the modified program's text segment is empty. The data segment is copied directly from the original program. The reassembly algorithm begins by placing unresolved constrained references at pinned addresses.

A *reference* is a link to data or instructions in a *dollop*. A dollop is a linear sequence of instructions linked by their fallthroughs. References are *unresolved* when they link to data or dollops in IR form. When a dollop is reconstructed from its IR into instructions and assigned a location in the modified program's address space, references are *resolved* to those particular addresses. In Figure 6, r_1, r_2 and r_3 are references. r_1 and r_2 are resolved and r_3 is unresolved.

A reference is *constrained* when there is a restriction on where its target may be placed within the modified program's address space. Because a reference includes an address (its target), the implementation of the reference itself must be at least as large as the encoding of that address. The size available for encoding the address may be limited when the address of two adjacent instructions are pinned.

In Figure 6, there is a 2-byte instruction i_1 at `0x400EE` and a 3-byte instruction i_2 at `0x400F0` and both have pinned addresses. The reference to the transformed instruction i_1' will have to encode the instruction's address when it is placed. No matter how the ISA encodes addresses (relative or absolute), the encoding cannot exceed two bytes without interfering with the reference at the adjacent pinned address. If the ISA does not support addressing the full address space in two bytes, the reference at `0x400EE` must be constrained.

Once a constrained unresolved dollop reference is placed at each pinned address, the reassembly algorithm determines which references can be unconstrained. Depending upon the ISA of the implementation target, there is a minimum size, *s*, necessary to store an instruction that addresses the entire address space. If the space between adjacent constrained unresolved references r_1 and r_2 is greater than *s*, our algorithm converts r_1 to an unconstrained unresolved reference. In Figure 6, r_2 would initially have been a constrained unresolved reference but has been converted to an unconstrained unresolved reference because there are no pinned addresses in [`0x4000F0`,`0x4000F0`+*s*).

For every remaining constrained unresolved reference, *r*, that references instruction *i*, a new unresolved unconstrained dollop reference, *r*', is added in the modified program's address space. *r* is resolved to *r*' through one or more intermediate references and *r*' is set to reference *i*. This is a process known as *chaining* [16]. In Figure 6, r_1 is a reference to d_1 that is chained through r_3.

At this stage of the reassembly, all unresolved references are unconstrained. Besides the information from the IRDB, the reassembly algorithm relies on three data structures:

- *uDR*: The list of unresolved references,
- *D*: A list of unplaced dollops (this list is initially empty),
- *M*: A mapping between instructions and their location in the modified program's address space.

The final stage of the reassembly algorithm is iterative: Every unresolved reference, r_u, to instruction *i* in list *uDR* is considered in turn until the list is empty.

Reference r_u is handled in one of two ways. Either *i* is already placed in the modified program's address space or it is not. In the former case, the reassembly algorithm simply emits a resolved unconstrained reference to *M*[*i*]. r_u is removed from *uDR* and the loop continues. The latter case is more involved — a dollop containing *i* must be retrieved or constructed and then placed. The reassembly algorithm searches *D* for *d*, the dollop containing *i*.

If no dollop is found, the reassembly algorithm constructs a dollop that contains *i*. The dollop construction process is straightforward. Dollop *d* begins with instruction i_0 and includes i_0's fallthrough i_1, i_1's fallthrough i_2, and so on. The last instruction in *d*, i_n, is the first instruction that has no fallthrough. The reassembly algorithm places the instructions of *d* linearly in a consecutive block of addresses. In Figure 6, d_2 is a placed dollop.

When there is no block of free space big enough to accommodate the instructions of *d*, the dollop may be split. Furthermore, large dollops may be split to fill small blocks of free space. Dollop *d* of instructions $i_1...i_s...i_n$ is split by choosing a split point, i_s. Dollop *d* is truncated to contain instructions $\{i_1...i_{s-1}\}$ and *d*' is built to contain instructions $\{i_s...i_n\}$. An unconstrained unresolved reference *r* that references i_s is appended to the end of *d*. The unresolved reference *r* is added to *uDR* and *d*' is added to *D*.

After *d* is placed, r_u is resolved and the map *M* is updated for all instructions in *d*. Any other unresolved references that target instructions in *d* are resolved as well. In Figure 6, r_2 is resolved to d_2.

An instruction i_1 in the just-placed dollop *d* may reference another instruction, i_2. If i_2 is already placed, that reference is resolved immediately. Otherwise, an unresolved reference r_2 is created that references i_2 and r_2 is added to *uDR*.

The modified program is completely reassembled when *uDR* is empty.

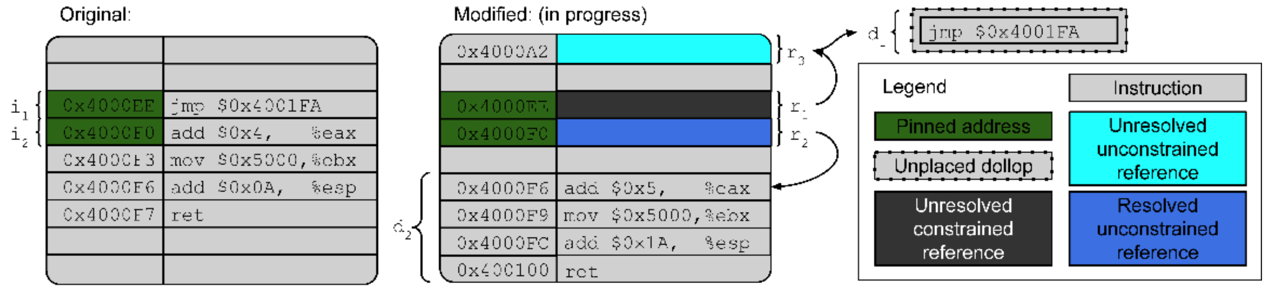


Figure 6: The reassembly algorithm in the process of reassembling a modified program.

Again, Figure 6 illustrates these concepts in the context of an program being reassembled. In this example we will describe Helix/Kevlar, the prototype implementation of our technique for the x86 ISA where there are variable length instructions. A jump — the implementation of references for this target — can be as short as two bytes (program control transfer is constrained to nearby locations) and as long as five (program control can be transferred anywhere in the program).

In this example there are two pinned addresses, two dollops and three references. Dollop d_2 is already placed; dollop d_1 is not. Reference r_1 began as a constrained unresolved reference to an instruction in dollop d_1. For expository purposes, assume that d_1 could not be placed at an address that is addressable in 2 bytes from r_1. Jump chaining was used and reference r_1 was resolved to r_3 and r_3 became an unresolved unconstrained reference to the instruction in d_1.

Because dollop d_2 is already placed, reference r_2 is resolved. Reference r_2 began as a constrained reference but because there were no pinned addresses in [`0x4000F0`,`0x4000F5`), r_2 was converted to an unconstrained reference.

We have evaluated Zipr's robustness and efficient using SPEC2006 on 32- and 64-bit machines. Zipr was able to properly transform every application in the benchmark suite. All quantitative results presented are from 32-bit executions of SPEC2006 on a test host with a quad-core 3.0GHz CPU (AMD Phenom II X4 B55) and 4GB of RAM that ran Ubuntu 10.04 LTS with version 4.4.3 of the GCC compiler suite. The baseline results for comparison were generated with a native run of SPEC2006 on that same host. All values presented in Figure 7 are normalized against those baseline results.

To quantify the overhead of the Helix/Kevlar technique itself, we compared the performance and size of the C-based SPEC applications before and after applying a user-specified Null Transformation. The Null Transformation is the most basic transformation. In fact, it is not a transformation at all. It is simply a no-op modification invoked on the IR during the User-specified Transformation stage of the Transformation Phase. In other words, the original and the modified programs are semantically equivalent. Therefore, any change in program size (on disk) or performance is a consequence of the rewriting algorithm *per se*.

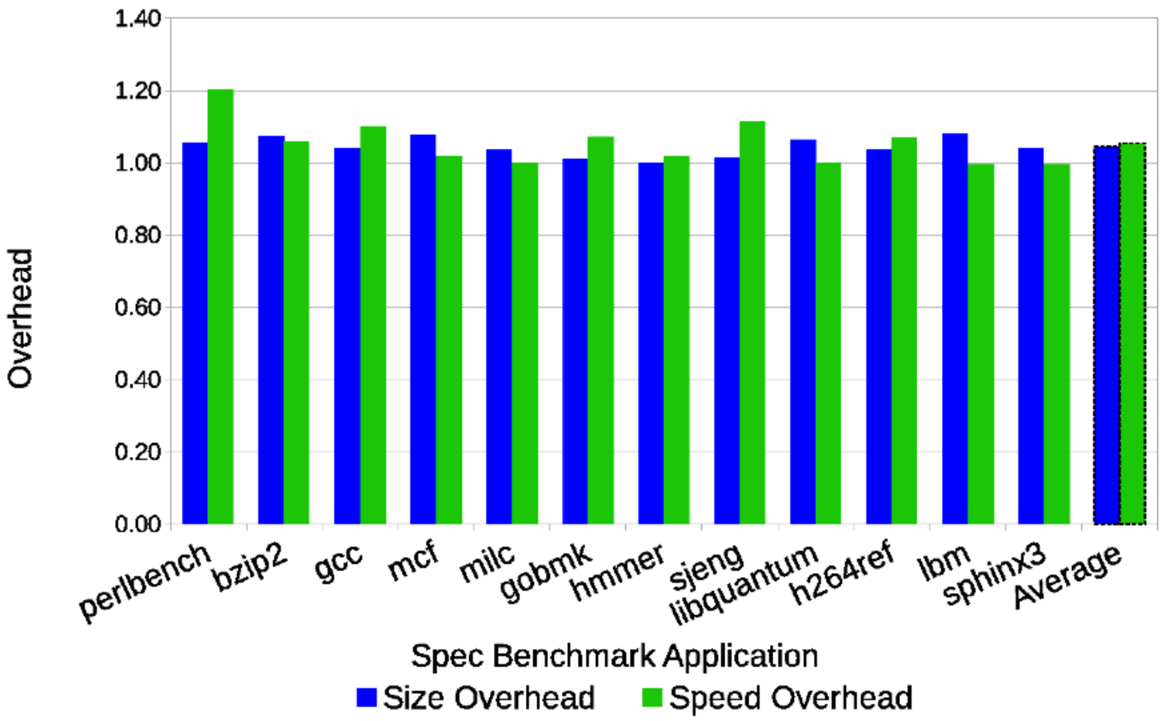


Figure 7: Inherent overhead of programs transformed using Helix/Kevlar prototype implementation.

The left bars of Figure 7 shows the size overhead for the C-based SPEC2006 benchmark applications when modified with the Null Transformation. On average, the modified binary programs are 4% larger than the original. The right bars of Figure 7 shows the performance overhead for each of the C-based SPEC2006 benchmark applications when modified with the Null Transformation. On average, the modified binary programs execute 5% slower than the original.

To reiterate, although we are presenting only the results for the C-based SPEC applications, Helix/Kevlar correctly transforms all of the SPEC2006 benchmarks whether they are implemented in C, C++ or Fortran. While there is ongoing work to improve the performance of the prototype implementation, the overall results prohibitively validate the feasibility and inherent efficiency of the rewriting technique.

Rodes et al. [21] describe an SLX program transformation (hereafter referred to as *P1*) that “[defends] binaries against intra-frame stack-based attacks, including overflows into local variables” even without access to the program’s source code. P1 “[applies] a combination of transformations, including variable reordering, random-sized padding between variables, and placement of canaries.”

P1 was initially implemented using Strata using the same API that Helix/Kevlar exposes to its users to rewrite programs statically. Helix/Kevlar applied P1 to a subset of the C-based SPEC2006 benchmark applications using the very same implementation. Therefore, the resulting statically rewritten programs benefitted from all the security afforded by P1 without the additional requirement of a runtime engine and without reimplementation.

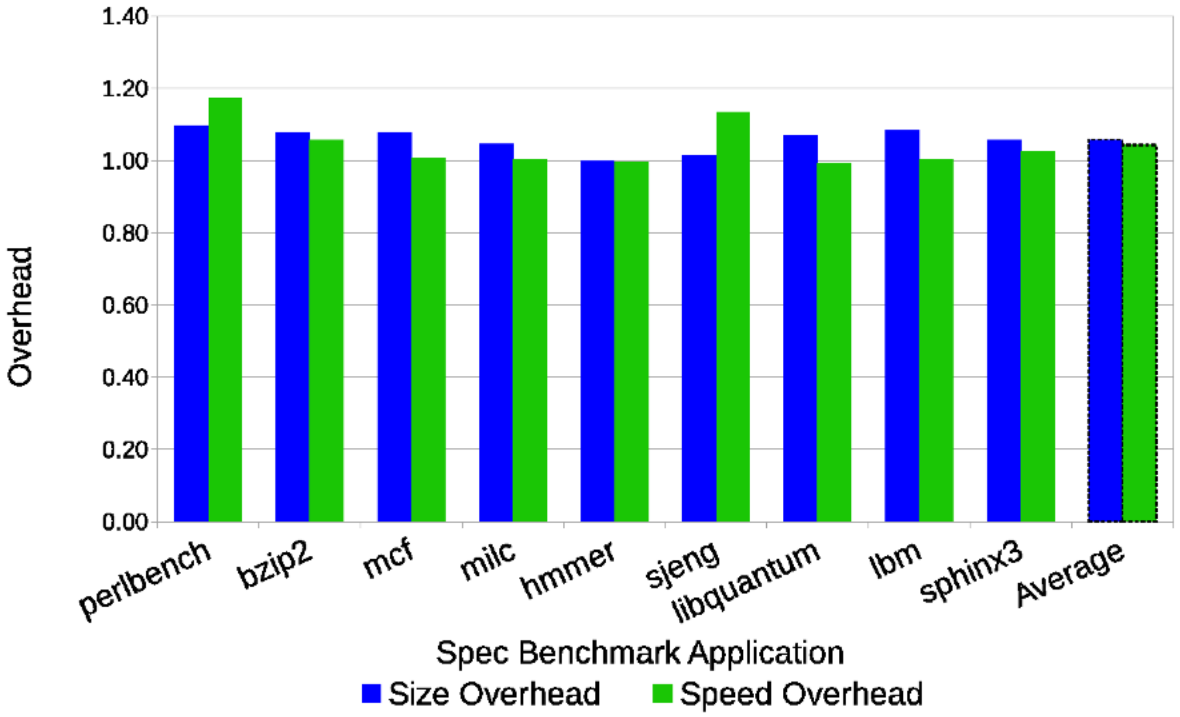


Figure 8: Overhead of P1.

The left bar of Figure 8 shows that the Helix/Kevlar methodology can add the security protection afforded by stack layout transformation with less than 6% increase in the program's on-disk size, on average. The right bar of Figure 8 shows that adding the security of P1 through the Helix/Kevlar methodology incurs less than a 5% performance penalty. Our experiments show GCC's built-in stack protection mechanism increases on-disk size by 4% and adds more than 5% to execution time on these same benchmarks, demonstrating Zipr's efficiency.

3.3 Helix/Kevlar Protections

The Helix/Kevlar architecture is flexible and powerful. It can dynamically apply a wide range of diversity transformations on a running binary, it can check and enforce various program properties that have been extracted from the binary or specified by an administrator, and it can insert remediation code. The following sections describe some of the Helix/Kevlar protections.

3.3.1 Instruction Location Transformation (ILX)

A powerful diversity technique is to randomize the location of code so an attacker has difficulty precisely locating targets of attack (e.g., entry point to functions, tables of pointers to functions, etc.). For example, most systems now routinely use Address Space Layout Randomization (ASLR) to make exploiting weaknesses difficult [35]. ASLR has several positive attributes. It is cheap to apply incurring little or no run-time overhead, and it can It can be applied to any binary, It is applied automatically—no user intervention or action is necessary.

Unfortunately, ASLR implementations have low entropy. ASLR on a 32-bit architecture only provides 16 bits of entropy. Furthermore, ASLR is not applied universally throughout the address space. Even when using dynamically-linked libraries, it is common for the main program text to start at a known fixed location. Because of these limitations, ASLR-protected code is subject to attack [5, 22, 27].

Instruction Location Transformation (ILX) is a technique that seeks to scatter instructions in a program randomly throughout the address space. In contrast to ASLR, ILX provides 31 bits of entropy on a 32-bit machine. Furthermore, ILX is applied universally to all segments. Thus, the major limitations of ASLR are eliminated.

Figure 9 conceptually illustrates ILX. The top left of the figure shows the control-flow graph of a particular program segment. The compiler and the linker collaborate to produce an executable file where instructions are laid out so they can be loaded into memory when the program is executed. A typical layout of code is shown at the bottom left of the figure. The right side of the figure shows the layout of the code when ILX is applied.

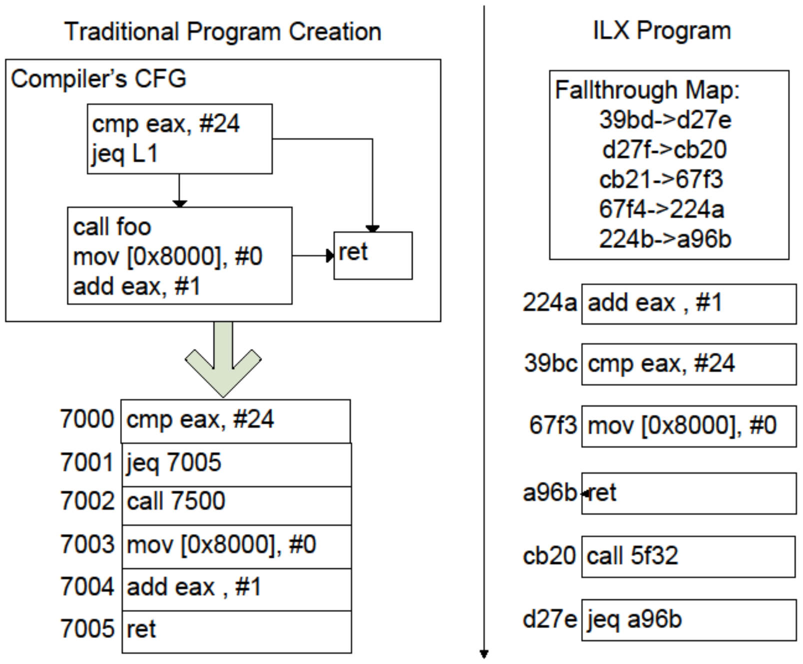


Figure 9: ILX code example

To link instructions together, an ILX Sprocket contains a fallthrough map shown at the top right of Figure 9. This map uses SPRI to specify the execution successor of each instruction in the program.

Together, we call the fallthrough map and randomized instruction locations an ILX Sprocket. Figure 10 shows how an ILX program could be created. First, STARS detects the instructions and functions in a program. Next, the program is analyzed for indirect branch targets, call sites, branches, etc. Finally, the reassembly engine uses this information to relocate the entire program with each instruction in a randomized location, and creates the fallthrough map.

To execute the randomized program, we use Strata to fetch and execute the instructions from the ILX Sprocket. Strata interprets the fallthrough map to fetch and execute instructions on the host hardware. Preliminary results indicate that this process can be made very efficient. The preliminary prototype achieved only 13% runtime overhead on the SPEC2006 benchmark suite. Furthermore, randomly scattering instructions throughout the address space significantly reduces the attack surface for mounting any arc-injection attacks, including attacks based on return-oriented programming techniques. Hiser et al. provides a more complete discussion of the benefits of ILX and full details of its implementation [11].

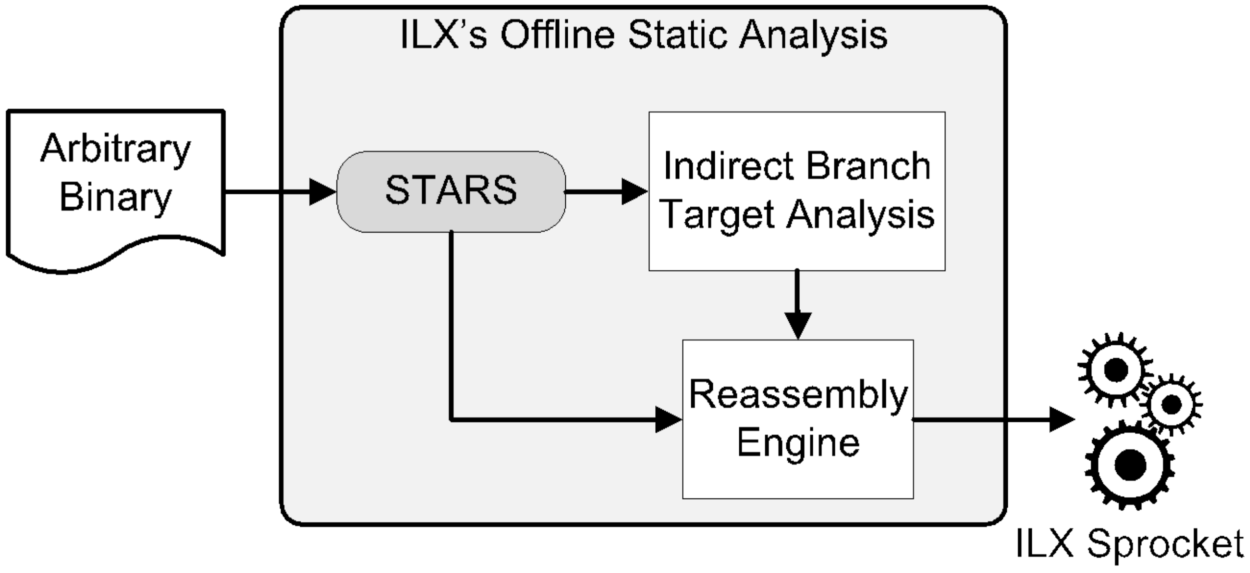

ILX's Offline Static Analysis
Arbitrary Binary
STARS
Indirect Branch Target Analysis
Reassembly Engine
ILX Sprocket

Figure 10: ILX Static Analysis

3.3.2 Stack Layout Transformation (SLX)

A common target of malicious attacks are locations on the stack (e.g., return addresses, frame pointers, function pointers, and critical data). SLX is a transformation that is applied to a running application to dynamically randomize the location of variables on the stack and place canaries to determine if an attack has been attempted.

Transformation of the stack frame layout for a function requires determination of:

1. The current layout of the stack frame, e.g. the addresses and sizes of various stack data objects (incoming arguments, saved registers, return address, local variables, outgoing arguments)
2. The instructions that generate an address of a data object on the run-time stack.

In principle, if this information were available, the layout of the stack frame could be changed and the instructions that generate stack addresses modified to reflect the new layout. The new layout of the stack frame could be based on any security-relevant criteria, e.g., memory objects could be placed in random order, padding introduced before, after or within the stack, canaries included, variables promoted to the heap, etc. While this information is readily available to the compiler when given a program in source code form, Helix/Kevlar must recover this information solely based on the binary representation.

In our approach, static analysis (STARS) is used to determine all the necessary details of the binary program. However, when starting with a binary program, precise determination of the stack layout and the instructions that generate stack addresses for any given function is problematic (indeed, even the very basic notion of a function is problematic at the binary level). Modern compilers employ a wide range of techniques to minimize both the use of storage and program execution time. The result is binary programs with unpredictable structures.

Our approach to determination of the stack layout and the instructions that reference the stack is based on two assumptions about addressing: (a) the predominant mechanism by which instructions access stack variables is through scaled or direct addressing based on an offset indicating the variable starting location, and (b) where indirect addressing is used, that use is for access to variables whose locations can be inferred from previous direct or scaled addressing. Starting with these assumptions, layout inferences are produced using a set of simple heuristics that rely upon additional assumptions concerning: (c) the manner in which the stack is allocated and deallocated, and (d) the general stack frame layout.

The assumptions listed above do not necessarily hold (although assumptions (c) and (d) hold for binaries produced by C/C++ compilers that use the `cdecl` x86 calling convention). Indeed through BED and TSET, the Helix/Kevlar architecture explicitly compensates for any transformations that might rely on erroneous information. Our approach to stack layout transformation is speculative. Initial inferences about the stack are created, and these inferences are then evaluated and refined if necessary to ensure that they preserve the program's semantics.

In our current approach, we limit transformations to placement of memory objects in random order and the introduction of random length padding. Vetting of these transformations is by testing with BED and TSET. Furthermore, we use diversity as an *error amplification* technique to detect bad stack layout inferences. The basic idea behind error amplification is as follows: if a hypothesized stack layout inference is correct, then any semantic-preserving transformations should result in a correct program variant. We can therefore develop a multitude of such transformations, e.g., permutation of the order of variables, and vet each of the variants. If any of them fail, and assuming that our transformation is correctly implemented, we can then not only reject the variant but also the inferred stack layout. Note that this process is the exact opposite of validating transformations by using testing to validate optimizing compiler transformations.

For each detected function in a binary, SLX randomizes the stack layout using an aggressive inference to reorder variables, e.g. using offsets in the disassembly of the program to infer variables. If the tests are passed, we use error amplification before creating the final variant. The layout is randomized a second time and the resulting program tested again. If the tests are passed following the second randomization, a third randomization is effected that reorders the stack elements and places padding between stack objects. If the tests are passed with this randomization, then the transformation is assumed to be satisfactory, and the analysis continues with the next function.

If one or more tests fail during analysis of a function, the inference about the stack layout is abandoned, and a simpler, less aggressive inference is used. The least aggressive inference besides not changing the function at all is one in which the entire stack frame is relocated but the order of variables is left unchanged. Preliminary work suggests that reordering variables is an effective error amplification technique as reordering misidentified variables will most likely result in a program crash. Thus, three rounds of error amplification appears sufficient to vet SLX transformations.

Our current approach has been evaluated on a variety of benchmarks, and the results are promising [21]. The use of BED and TSET resulted in binaries whose functions were transformed with different levels of aggressiveness, ranging from no transforms, to transforms that reordered a subset of the local variables, and in some cases, we were able to infer and reorder all local variables on the stack frame. The ability to reorder stack variables for security purposes is standard in some compilers, e.g., the ProPolice extension to gcc reorders buffers higher in memory than other variables to prevent local overflows [7]. Our results demonstrate that we can enable similar transformations but using only binaries.

3.3.3 Heap Randomization and Transformation (HLX)

Helix/Kevlar's Heap Layout Transformation (HLX) provides protection against a variety of common memory errors, such as buffer-overflows, use-after-free, and double-free errors. It achieves these protections by detecting (using STARS analysis) memory allocations within the program and rewriting the allocations to randomly increase the allocation size. HLX also detects memory deallocation sites and maintains a pool of objects that were marked as free. When additional memory is needed (such as the free object pool has become too large), the free pool is checked and objects are randomly selected for deallocation.

Table 1 provides an example. The left portion shows unprotected source code that allocates a buffer, and uses that buffer to manipulate input. Unfortunately, the code has a off-by-one error, and allocates too few bytes to hold the newly formed string, perhaps because additional characters were were added to the manipulation, but the size of the buffer was not updated. The right side of the figure shows how HLX would transform the program. The amount of memory allocated gets increased by a random amount, and free pool management code is inserted. In this case, the off-by-one error is converted from a possibly crash-inducing bug, into a fully-correct program. While not all programs can be completely repaired, the transform still prevents exploits because an attacker cannot reliably predict where heap items may be located, or what size a buffer might be to predictably overrun the buffer.

Table 1: Example without and with HLX, respectively

`int size = strlen(input);`	`int size = strlen(input);`
`char* newValue = malloc(size+1);`	`cleanup_free_pool();`
`strcpy(newValue,input);`	`char* newValue =`
`sprintf(newValue, "%s!\n", input);`	`malloc(random_increase(size+1));`
`log("The input is %s", newValue);`	`strcpy(newValue,input);`
`free(newValue);`	`sprintf(newValue, "%s!\n", input);`
	`log("The input is %s", newValue);`
	`add_to_free_pool(newValue);`

Unlike the example, however, HLX provides high-entropy randomization on a binary program where no source code is available, like the rest of Helix/Kevlar.

3.3.4 Instruction Set Randomization (ISR)

A common and very dangerous form of security attack involves exploiting a vulnerability to inject malicious code into an executing application and then cause the injected code to be executed. A theoretically-strong approach to defending against any type of code-injection attack (irrespective of the vulnerability) is to create and use a process-specific instruction set that is created by a randomization algorithm. Code injected by an attacker who does not know the randomization key will be invalid for the randomized processor effectively thwarting the attack.

Helix/Kevlar takes advantage of ISR to help defeat these kinds of attacks. ISR uses Helix/Kevlar's static analysis and runtime support to identify code locations, and encrypt them during a process' loading procedure. Helix/Kevlar versions of ISR is based on the first practical version of ISR [12]. Helix/Kevlar further extends this technology to monitor the application for dynamically loaded code (shared objects or .dll's) and encrypts that code as it enters the runtime system.

Helix/Kevlar's code-injection security can be further enhanced by configuring it as a *metamorphic shield* [18]. The metamorphic shield (MMS) technology not only randomizes the code at program startup, but periodically re-randomizes the code's encryption characteristics as the program is running. Such randomization prevents attackers from exhaustively searching for the encryption key, keeping the program safe from code injection from even the most determined attackers.

3.3.5 PC Confinement (PCC)

ISR provides diversity which prevents malicious code from being injected into the running application, but an attacker may still be able to re-use code that is already in the application to enact security violations, called an arc-injection attack [20]. Indirect branches, such as function calls via function pointers and function return instructions, are vulnerable to such an attack if the branch's data is overwritten with a buffer overflow, format string issue, or other program weakness. Table 2 contains an example of a possible arc-injection attack.

Table 2: Example of arc-injection attack

```
void main(){
   auth = authenticate();
   vulnerable_code();
   if(auth) {
      send_secret_data();
   }
}
```

In the table, if the code in `vulnerable_code()` can overwrite the function's return address (or even just part of the function's return address via a partial overwriting attack! [1]), the return instruction can possibly jump anywhere in the program. It may jump to the system() function to execute shell commands, or to the send_secret_data() call, to more steathily violate the application's security policy.

Helix/Kevlar's ILX feature can defeat many of these attacks by randomizing the application's code. However, Strata's translation and Sprocket execution code are at static locations, which may still be targets of arc-injection attacks. Helix/Kevlar can protect all statically located code by employing *PC confinement* (PCC). PCC is a type of program shepherding where indirect branches are monitored and only allowed to transfer control to ILX-randomized code [14].

Helix/Kevlar's static analysis identifies the location of valid indirect control transfers, and the run-time environment efficiently monitors the execution of indirect branches. Indirect control flow is only allowed if the destination is acceptable. Consequently, PCC and ILX can disallow control transfers to unrandomized code, such as Strata's Sprocket execution code, thereby eliminating the vast majority of arc-injection attacks.

3.4 Technology Communication

As part of our approach to transitioning the Helix/Kevlar technology, we participated in a variety of meetings, prepared various publications, and gave several presentations during this period. Our most significant and visible communications are:

- At the request of Patrick Hurley of AFRL, we provided a demonstration copy of Helix/Kevlar to members of the Technical Cooperation Program (TTCP). The partners include Australia, New Zealand, Canada, United Kingdom, and the United States.
- We presented a paper, *Security Protection of Binary Programs,* at the 10th IET System Safety and Cyber-Security Conference in Bristol, UK on October 21, 2015 through October 22, 2015.
- Jack Davidson presented a briefing on Helix technology to the United States Postal Service (USPS) on October 16, 2015.
- Jack Davidson was an invited participant in the Moving Target Workshop held at George Mason University on August 31, 2015 through September 1, 2015. The title of his talk was "Evaluating the Effectiveness of the Helix's Metamorphic Shield."
- Anh Nguyen-Tuong met with Azbil, a maker of industrial automation and control products, to discuss the use of Helix/Kevlar to protect industrial systems.
- Jack Davidson made a presentation on cyber security research to the Department of Computer Science's Industrial Advisory Board on July 16, 2015. Representatives of Capital One, Palantir, Appian, Lockheed Martin, and Excella were in attendance.
- We presented a paper, *Joza: Hybrid Taint Inference for Defeating Web Application SQL Injection Attacks*, at the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks in Rio de Janeiro, Brazil on June 22–25, 2015.
- Jack Davidson presented a technology briefing to Northrop Grumman on May 28, 2015.
- Anh Nguyen-Tuong Airbus presented a briefing on Helix technology to Airbus cyber security group on October 19, 2015.

4.0 RESULTS AND DISCUSSION

We present our results (as indicated in the following subsections) for each reporting period during the project.

4.1 Period 1: 19-FEB-2015 through 30-JUN-2015

4.1.1 Progress Against Planned Objectives

A major objective of this effort is to demonstrate the technical readiness of portions of the Helix technology developed under contract FA8650-10-C-7025, FA8650-13-2-0096, and others. Towards this goal, we provided Northrop Grumman with a virtual machine image that contained the complete Helix tool chain, directions for applying the tools to applications, and sample programs that had already been protected. We worked with the Northop Grumman personnel (Russ Hall and Kevin Reynolds) to enable Northrop Grumman to perform red team attacks against protected applications to demonstrate the effectiveness of the Helix protections. We expect the results of this exercise in the near future (i.e., a few weeks).

4.1.2 Technical Accomplishments this Period

There were several major technical accomplishments this period. First, we completed the retargeting of Strata, our software dynamic translator, to Windows (64-bit). Strata is a key component of Helix in that it provides the capability to dynamically translate code. This capability provides the ability to apply various diversity transformations and to shift the attack surface if desired. Akshay Joshi (`asj5b@virginia.edu`) is doing this work as part of his Ph.D. research.

A second major technical accomplishment was to use Zipr, our static binary rewriting technology, to rewrite, diversify and protect several interpreters/JIT such as the main executable of Java VM, Python, and `Node.js`, the Javascript runtime).

A third major technical accomplishment was to demonstrate that Zipr was able to apply and compose diversity transformations (i.e., block-level location randomization and stack padding) to a small JIT program.

A fourth major accomplishment was a demonstration that Zipr with a moving target defense could defeat blind ROP attacks. A paper is being written for submission to a major security conference. Will Hawkins (`whh8b@virginia.edu`) is doing this work as part of his Ph.D. research.

4.1.3 Improvements to Prototypes This Period

We enhanced Zipr to handle the main executable of the Java VM and other interpreters such as Python and Javascript.

We continued to fix both software errors and performance and scalability issues throughout the Helix tool chain (e.g., STARS, Strata, Zipr, etc.)

4.2 Period 2: 01-JUL-2015 through 15-AUG-2015

4.2.1 Progress Against Planned Objectives

A major objective of this effort is to demonstrate the technical readiness of portions of the Helix technology developed under contract FA8650-10-C-7025, FA8650-13-2-0096, and others. Towards this end, we have been improving our technology to work on the Java VM (JVM) and the Windows 7 platform.

Our focus with Java has been on the Ubuntu platform, with ultimate goals of using this for GCCS under Solaris. To achieve our goal, we’ve been using Zephyr’s static binary rewriter, Zipr, to effect changes in binary programs. Zipr’s main benefit is that it is interoperable with dynamic code generation (or just in time compilation also known as JITting). We have block-level ILR working on the JVM core components, and initial prototypes working on Solaris with smaller programs.

On Windows, our core analysis and dynamic transformation engines are working with all major program features (such as threads and exception handling).

4.2.2 Technical Accomplishments this Period

There were several major technical accomplishments this period. First, we enhanced Strata, our software dynamic translator, to Windows (64-bit). Strata is a key component of Helix in that it provides the capability to dynamically translate code. This capability provides the ability to apply various diversity transformations and to shift the attack surface if desired. Akshay Joshi (`asj5b@virginia.edu`) is doing this work as part of his Ph.D. research. Another student, Jian Xiang (jx5c@virginia.edu), is working on benchmarking Strata under Windows to identify and fix performance issues.

A second major technical accomplishment was to use Zipr, our static binary rewriting technology, to rewrite, diversify the core shared libraries of the Java virtual machine. A major component of the JVM is `libjli.so`, and we have achieved block-level ILR protection. Block-level ILR relocates all of the basic blocks in a program so that each machine can have a randomized binary to help break the software monoculture. This diversity transform helps defend against attacks that rely on knowing the location of key code sections. To ensure that performance overheads are acceptable, we have been working on benchmarking Zipr on Ubuntu with the help of a student, Will Hawkins (`whh8b@virginia.edu`).

We have further improved our development infrastructure so that Ubuntu, Windows, and Solaris versions all integrated into one version of the source. This integration allows bug fixes to immediately be applied to all versions of the software. Further our testing infrastructure now includes nightly tests for Zipr, and we are working to include Windows and Solaris testing nightly.

4.2.3 Improvements to Prototypes This Period

On Ubuntu platforms, we enhanced Zipr to handle shared objects, including the core shared objects of the Java VM. The core of the JVM is a 12MB shared object called `libjli.so`. Libjli makes up over 90% of the JVM's functionality. We have tested the JVM against jedit, a production-quality, GUI-based, full-featured text editor. Full functionality is retained. Initial work on Zipr for Solaris is promising. We have been able to rewrite small executables successfully.

We further continued development on the Windows platform. The dynamic translation execution engine now supports all major features of Windows executables (e.g., threads, exception handling, signal delivery, etc.) Further, the analysis engine is now capable of analyzing Windows PE files, creating the IRDB representation of the program, and producing rewrite results that can be used by either static or dynamic binary rewriters.

We continued to fix both software errors and performance and scalability issues throughout the Helix/Kevlar tool chain (e.g., STARS, Strata, Zipr, etc.)

4.3 Period 3: 16-AUG-2015 through 30-SEP-2015

4.3.1 Progress Against Planned Objectives

A major objective of this effort is to demonstrate the technical readiness of portions of the Helix technology developed under contract FA8650-10-C-7025, FA8650-13-2-0096 and others. During this period, we focused on robustness of our static analysis and binary rewriting technology.

4.3.2 Technical Accomplishments this Period

We leveraged a Red Team exercise performed by Raytheon for another project to test Helix/Kevlar's ability to process the Global Positioning Navigation and Timing Systems (GPNTS) being developed by Raytheon for the Navy.

The major technical accomplishments this period are:

- Retargeted Helix/Kevlar to RedHat Enterprise Linux.
- Refactored the STARS interfaces so that analysis could be done on transformed binaries. This change enables easy composition of defenses.
- Worked on moving Helix/Kevlar to the Cloud to enable users to apply Helix/Kevlar protections through an easy-to-use web interface. It also allows the use of high-end, scalable computing resources for various tasks.

4.3.3 Improvements to Prototypes This Period

We have made numerous improvements to the Helix/Kevlar toolchain, in particular for x86-64. These include:

- Ability to run Helix/Kevlar on Redhat Enterprise Linux (RHEL) and process RHEL binaries.
- Fixed numerous bugs in STARS that were exposed through the Raytheon Red Team exercise.

In addition we continue to fix bugs we encounter as part of the porting process to x86-64. The net result is to vastly improve the robustness of Kevlar across both the x86-32 and the x86-64 bit versions.

4.4 Results Discussion

During the period of the project, we have done much to understand and promote the possible transition of Helix/Kevlar. It is important to note that Helix/Kevlar is playing a key role in two new efforts. First, Helix/Kevlar is a key component of our entry in DARPA's Cyber Grand Challenge. We were one of seven (out of 104) teams that have advanced to the finals to be held at Def Con 24 to be held in Las Vegas August 4–7, 2016.

Second, Helix/Kevlar is also a key component of our effort within DARPA's Cyber Fault-tolerant Attack Recovery (CFAR) program. This project seeks to build *N*-variant systems that are provable able to withstand attack. Helix/Kevlar's diversity transforms are heavily used.

Third, various meetings, publications and presentations have helped us connect with possible transition parters and taught us the constraints customers may have for a transitionable technology. Our porting to Windows, Solaris and RedHat platforms has helped us both gain an deeper understanding these issues and constraints, as well as made our prototype technology more attractive to potential parters. We have learned much, namely that different operating systems have different default compilers, which can provide significant challenges to a tool that works on the compiler's output (a binary program).

5.0 CONCLUSIONS

Security weaknesses in DoD information systems remain a major challenge for system stakeholders. We have advanced the transition of technology developed under the Helix and PEASOUP projects to protect Air Force systems of interests. The result are expected to be an asset that, if widely deployed by the DoD, would enable a high level of confidence in the security of DoD systems, in particular, confidence that certain classes of critical vulnerabilities were no longer subject to possible exploitation.

We have leveraged the opportunity to take the Helix architecture one step closer to deployment in real systems by developing Helix/Kevlar, a completely automatic system for securing applications against attack by well-funded, determined malicious adversaries. Helix/Kevlar armors binary programs and protects them from attacks which could arise from the inevitable vulnerabilities that remain after deployment. The source code is not required nor are any other development artifacts. These features make Helix/Kevlar of particular value for software systems that have to be used but for which no development information is available.

During this project we have done much to understand and promote the possible transition of Helix/Kevlar. First, various meetings, publications and presentations have helped us connect with possible transition partners and taught us the constraints customers may have for a transitionable technology. Our porting to Windows and Solaris platforms has helped us both gain a deeper understanding these issues and constraints, as well as made our prototype technology more attractive to potential partners. We have learned much, namely that different operating systems have different default compilers, which can provide significant challenges to a tool that works on the compiler's output (a binary program). In particular, some of our Linux-based tools assumed a particular calling convention, and different systems use different calling conventions. Abstracting the calling convention as much as possible eases technology transition.

6.0 REFERENCES

1. S. Alexander. Defeating compiler-level buffer overflow protection. *J-LOGIN*, 30(3):59–71, 2005.
2. Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 295–308, New York, NY, USA, 2013. ACM.
3. Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.
4. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, October 1991.
5. Tyler Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 0x0b(0x3b), 2002.
6. Andrew Edwards, Hoi Vo, Amitabh Srivastava, and Amitabh Srivastava. Vulcan binary transformation in a distributed environment. Technical report, Microsoft, 2001.
7. H. Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice), 2003.
8. William H. Hawkins, Jason D. Hiser, and Jack W. Davidson. Dynamic canary randomization for improved software security. In *To appear in Proceedings of the the 11th Annual Cyber and Information Security Research Conference*, CISR'16, New York, NY, USA, 2016. ACM.
9. Hex-Rays. IDA Pro. `http://www.hex-rays.com/products/ida/index.shtml`.
10. Jason Hiser, Clark L. Coleman, Michele Co, and Jack W. Davidson. MEDS: The memory error detection system. In Fabio Massacci, Samuel T. Redwine Jr., and Nicola Zannone, editors, *Proceedings of the First International Symposium on Engineering Secure Software and Systems ESSoS*, volume 5429 of *Lecture Notes in Computer Science*, pages 164–179. Springer, 2009.
11. Jason D. Hiser, Anh Nguyen-Tuong, Michele Co, Jack W. Davidson, and Matthew Hall. ILR: Where'd my gadgets go? *IEEE Symposium on Security & Privacy*, pages 571–585, May 2012.
12. Wei Hu, Jason Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the Second International Conference on Virtual Execution Environments*, pages 2–12, Ottawa, Canada, June 2006. ACM Press.
13. Wei Hu, Jason D. Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd*

International Conference on Virtual Execution Environments, pages 2–12. ACM Press New York, NY, USA, 2006.

14. Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Eleventh USENIX Security Symposium*, august 2002.
15. James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI ’95, pages 291–300, New York, NY, USA, 1995. ACM.
16. Bruce W. Leverett and Thomas G. Szymanski. Chaining span-dependent jump instructions. *ACM Trans. Program. Lang. Syst.*, 2(3):274–289, July 1980.
17. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’05, pages 190–200, New York, NY, USA, 2005. ACM.
18. A. Nguyen-Tuong, A. Wang, J.D. Hiser, J.C. Knight, and J.W. Davidson. On the effectiveness of the metamorphic shield. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 170–174. ACM, 2010.
19. Mathias Payer and Thomas R. Gross. Generating low-overhead dynamic binary translators. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR ’10, pages 22:1–22:14, New York, NY, USA, 2010. ACM.
20. J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *Security & Privacy, IEEE*, 2(4):20–27, 2004.
21. Benjamin Rodes, Anh Nguyen-Tuong, Jason D. Hiser, John C. Knight, Jack W. Davidson, and Michele Co. Against stack-based attacks using speculative stack layout transformation. In *Proceedings of the Third International Conference on Runtime Verification*, RV’12, pages 308–313, Berlin, Heidelberg, September 2012. Springer-Verlag.
22. G.F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib (c). In *2009 Annual Computer Security Applications Conference*, pages 60–69. IEEE, 2009.
23. Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop*, volume 1997, pages 1–8, 1997.
24. B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *WCRE ’02: Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 45–54, 2002.
25. K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO ’03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.
26. Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation*, September 2001.

27. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS ’04, pages 298–307, New York, NY, USA, 2004. ACM.

28. Matthew Smithson, Kapil Anand, Aparna Kotha, Khaled Elwazeer, Nathan Giles, and Rajeev Barua. Binary rewriting without relocation information. *University of Maryland, Tech. Rep*, 2010.

29. Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. HDTrans: a low-overhead dynamic translator. *SIGARCH Computer Architecture News*, 35:135–140, March 2007.

30. Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI ’94, pages 196–205, New York, NY, USA, 1994. ACM.

31. Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

32. Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, pages 7–12. IEEE, 2005.

33. Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC ’12, pages 299–308, New York, NY, USA, 2012. ACM.

34. Daniel. Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. IEEE Security & Privacy, 7(1):26–33, Jan.-Feb. 2009.

35. Jun Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent runtime randomization for security. In Proceedings of the 22nd International Symposium on Reliable Distributed Systems, pages 260–269, oct. 2003.

36. Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE ’14, pages 129–140, New York, NY, USA, 2014. ACM.

37. Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automatic profiling and optimization. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, SOSP ’97, pages 15–26, New York, NY, USA, 1997. ACM.

LIST OF SYMBOLS, ABREVIATIONS AND ACRONYMS

DoD	Department of Defense
PEASOUP	Preventing Exploits Against Software of Uncertain Provenance
DARPA	Defense Advanced Research Projects Agency
IARPA	Intelligence Advanced Research Projects Agency
NSF	National Science Foundation
VM	Virtual Machine
IET	Institution of Engineering and Technology
IEEE	Institute of Electrical and Electronics Engineers
ROP	Return-oriented programming
IFIP	International Federation for Information Processing
IRDB	Intermediate Representation Database
SDT	software dynamic translator
TRL	Technical Readiness Level
STARS	STatic Analysis of Reactive Systems
BED	Behavior Equivalence Detection

TSET	Test Suite Evaluation Technology
IR	Intermediate Representation
SSA	Static Single Assignment
SPRI	Sprocket Program Rewriting Interface
ILX	Instruction Location Transformation
API	Application Program Interface
PC	Program Counter
IB	Indirect Branches
IBT	IB Target
LLV	Low Level Virtual Machine
RAM	Random Access Memory
CPU	Central Processing Unit
GB	Gigabyte
SLX	Stack Layout Transformation
ILX	Instruction Location Transformation
ASLR	Address Space Layout Randomization
BED	Behavior Equivalence Detection
HLX	Heap Layout Transformation
PCC	PC Confinement
TTCP	Technical Cooperation Program
USPS	United States Postal Service
GCCS	Global Command and Control System
JVM	Java Virtual Machine
GUI	Graphical User Interface
GPNTS	Global Positioning Navigation and Timing Systems
RHEL	Redhat Enterprise Linux